CHAPTER 7
Statistical Machine Learning

Introduction

Statistical **Machine Learning (ML)** is a branch of **Artificial Intelligence (AI)** that combines statistics and computer science to create models that can learn from data and make predictions or decisions. Statistical machine learning has many applications in fields as diverse as computer vision, speech recognition, bioinformatics, and more.

There are two main types of learning problems: supervised and unsupervised learning. Supervised learning involves learning a function that maps inputs to outputs, based on a set of labeled examples. Unsupervised learning involves discovering patterns or structure in unlabeled data, such as clustering, dimensionality reduction, or generative modeling. Evaluating the performance and generalization of different machine learning models is also important. This can be done using methods such as cross-validation, bias-variance tradeoff, and learning curves. And sometimes when supervised and unsupervised are not useful semi and self-supervised techniques may be useful. This chapters cover

only supervised machine learning, semi-supervised and self-supervised learning. Topics covered in this chapter are listed in the *Structure* section below.

Structure

In this chapter, we will discuss the following topics:

- Machine learning
- Supervised learning
- Model selection and evaluation
- Semi-supervised and self-supervised leanings
- Semi-supervised techniques
- Self-supervised techniques

Objectives

By the end of this chapter, readers will be introduced to the concept of machine learning, its types, and the topic associated with supervised machine learning with simple examples and tutorials. At the end of this chapter, you will have a solid understanding of the principles and methods of statistical supervised machine learning and be able to apply and evaluate them to various real-world problems.

Machine learning

ML is a prevalent form of AI. It powers many of the digital goods and services we use daily. Algorithms trained on data sets create models that enable machines to perform tasks that would otherwise only be possible for humans. Deep learning is also popular subbranch of machine learning that uses neural networks with multiple layers. Facebook uses machine learning to suggest friends, pages, groups, and events based on your activities, interests, and preferences. Additionally, it employs machine learning to detect and

remove harmful content, such as hate speech, misinformation, and spam. Amazon, on the other hand, utilizes machine learning to analyze your browsing history, purchase history, ratings, reviews, and other factors to suggest products that may interest or benefit you. In healthcare it is used to detect cancer, diabetes, heart disease, and other conditions from medical images, blood tests, and other data sources. It can also monitor patient health, predict outcomes, and suggest optimal treatments and many more. Types of learning include supervised, unsupervised, reinforcement, self-supervised, and semi-supervised.

Understanding machine learning

ML allows computers to learn from data and do things that humans can do, such as recognize faces, play games, or translate languages. As mentioned above, it uses special rules called algorithms that can find patterns in the data and use them to make predictions or decisions. For example, if you want to teach a computer to recognize cats, provide it with numerous pictures of cats and other animals, and indicate which ones are cats and which ones are not. The computer will use an algorithm to learn the distinguishing features of a cat, such as the shape of its ears, eyes, nose, and whiskers. When presented with a new image, it can use the learned features to determine if it is a cat or not. This is how machine learning works.

ML is an exciting field that has enabled us to accomplish incredible feats, such as identifying faces in a swimming pool or teaching robots new skills. It is an intelligent technology that learns from data, allowing it to improve every day, from playing games of darts to driving on the highway. It is also a source of inspiration, encouraging curiosity and creativity, whether it's drawing a smiling sun or writing a descriptive poem. Additionally, many of us are

familiar with ChatGPT, which is also powered by data, statistics, and machine learning.

Role of data, algorithm, statistics

Data, algorithms, and statistics are the three main components of machine learning. And as we know about these. Let us try to understand their roles with an example. Suppose we want to create a machine learning model that can classify emails as spam or not spam. The role of data here is that first we need a dataset of emails that are labeled as spam or not spam. This is our data. Then we need to choose an algorithm that can learn from the labeled data and predict the labels for new emails. This can be a supervised algorithm like logistic regression, decision tree, or neural network. This is our algorithm. Along with these two, we need to use statistics to evaluate the performance of our algorithm on the data. We can use metrics such as accuracy, precision, recall, or F1 score to measure how well our algorithm can classify emails as spam or not spam. We can also use statistics to tune the parameters of our algorithm, such as the learning rate, the number of layers, or the activation function. These are our statistics. This is how data, algorithm, and statistics play a role in machine learning. We further discuss this a lot in this chapter with tutorials and examples.

Inference, prediction and fitting models to data

ML has two common applications: inference and prediction. These require different approaches and considerations. It is important to note that inference and prediction are two different goals of machine learning. Inference involves using a model to learn about the relationship between input and output variables. It includes the effect of each feature on the outcome, the uncertainty of the estimates, or the causal mechanisms behind the data. Prediction involves using a

model to forecast the output for new or unseen input data. This can include determining the probability of an event, classifying an image, or recommending a product.

Fitting models to data is a general process that applies to both inference and prediction. The specific approach can vary depending on the problem and data. By fitting models to data, we can identify the best model to represent the data and perform the desired task, whether it be inference or prediction. Fitting models to data involves choosing the type of model, the parameters of the model, the evaluation metrics, and the validation methods.

Supervised learning

Supervised learning uses labeled data sets to train algorithms to classify data or predict outcomes accurately. For example, using labeled data of dogs and cats to train a model to classify them, sentiment analysis, hospital readmission prediction, spam email filtering.

Fitting models to independent data

Fitting models to independent data involves data points that are not related to each other. The model does not consider any correlation or dependency between them. For example, when fitting a linear regression model to the height and weight of different people, we can assume that one person's height and weight are independent of another person. Fitting models to independent data is more common and easier than fitting models to dependent data. Another example is, suppose you want to find out how the number of study hours affects test scores. You collect data from 10 students and record how many hours they studied and what score they got on the test. You want to fit a model that can predict the test score based on the number of hours studied. This is an example of fitting models to independent data, because one student's hours and test score are not related

to another student's hours and test score. You can assume that each student is different and has his or her own study habits and abilities.

Tutorial 7.1: To implement and illustrate the concept of fitting models to independent data, is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
# Define the data
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) # Number of h
ours studied
y = np.array([50, 60, 65, 70, 75, 80, 85, 90, 95, 100]) #
Test score
# Fit the linear regression model
m, b = np.polyfit(x, y, 1) # Find the slope and the interc
ept
# Print the results
print(f"The slope of the line is {m:.2f}")
print(f"The intercept of the line is {b:.2f}")
print(f"The equation of the line is y = {m:.2f}x + {b:.2f}")
# Plot the data and the line
# Data represent the actual values of the number of hou
rs studied and the test score for each student
# Line represents the linear regression model that predi
cts the test score based on the number of hours studied
plt.scatter(x, y, color="blue", label="Data") # Plot the d
ata points
plt.plot(x, m*x + b, color="red", label="Linear regressio
n model") # Plot the line
plt.xlabel("Number of hours studied") # Label the x-axis
plt.ylabel("Test score") # Label the y-axis
plt.legend() # Show the legend
plt.savefig('fitting_models_to_independent_data.jpg',dpi
=600,bbox_inches='tight') # Show the figure
```

```
21. plt.show() # Show the plot
```

Output:

```
1. The slope of the line is 5.27
2. The intercept of the line is 48.00
3. The equation of the line is y = 5.27x + 48.00
```

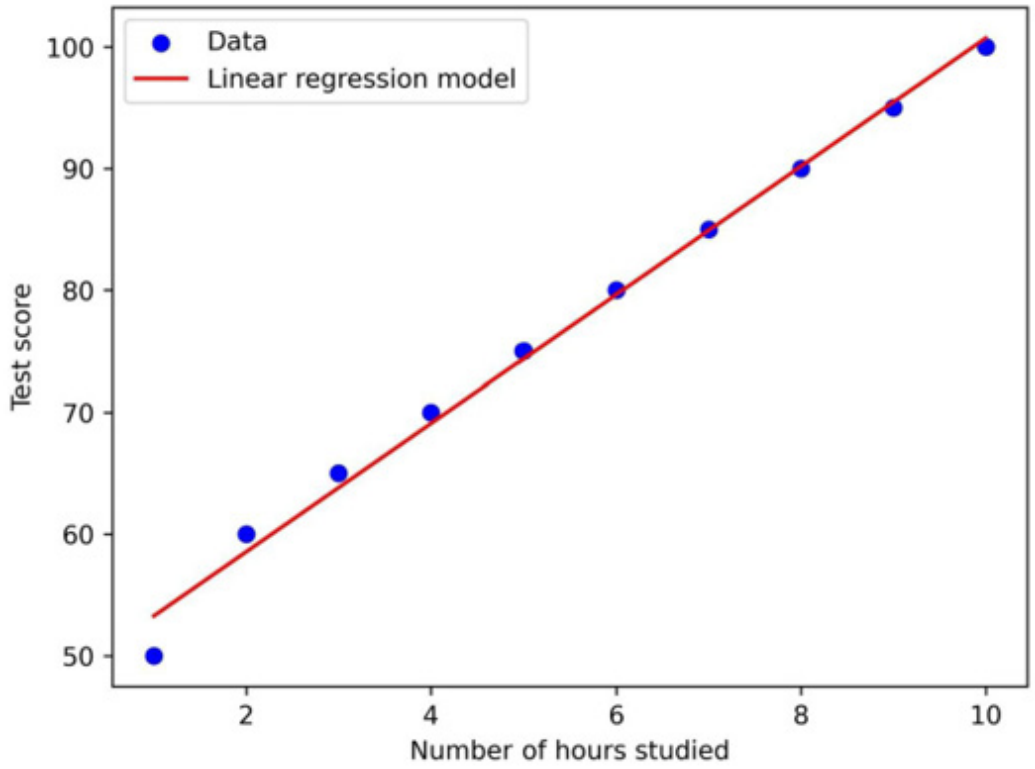

Figure 7.1: *Plot fitting number of hours studies and test score*

In *Figure 7.1*, the data (dots) points represent the actual values of the number of hours studied and the test score for each student and the red line represents the fitted linear regression model that predicts the test score based on the number of hours studied. *Figure 7.1* shows that the line fits the data well and that the student's test score increases by almost five points for every hour they study. The line also predicts that if students did not study at all, their score would be around 45.

Linear regression

Linear regression uses linear models to predict the target variable based on the input characteristics. A linear model is a mathematical function that assumes a linear relationship between the variables, meaning that the output can be expressed as a weighted sum of the inputs plus a constant term. For example, a linear model could be used to predict the price of a house based on its size and location can be represented as follows:

$$price = w1 * size + w2 * location + b$$

Where *w1* and *w2* are the weights or coefficients that measure the influence of each feature on the price, and b is the bias or intercept that represents the base price.

Before moving to the tutorials let us look at the syntax for implementing linear regression with **sklearn**, which is as follows:

```
# Import linear regression
from sklearn.linear_model import LinearRegression
# Create a linear regression model
linear_regression = LinearRegression()
# Train the model
linear_regression.fit(X_train, y_train)
```

Tutorial 7.2: To implement and illustrate the concept of linear regression models to fit a model to predict house price based on size and location as in the example above, is as follows:

```
# Import the sklearn linear regression library
import sklearn.linear_model as lm
# Create some fake data
x = [[50, 1], [60, 2], [70, 3], [80, 4], [90, 5]]
# Size and location of the houses
y = [100, 120, 140, 160, 180] # Price of the houses
# Create a linear regression model
```

```
model = lm.LinearRegression()
# Fit the model to the data
model.fit(x, y)
# Print the intercept (b) and the slope (w1 and w2)
print(f"Intercept: {model.intercept_}") # b
print(f"Coefficient/Slope: {model.coef_}") # w1 and w2
# Predict the price of a house with size 75 and location 3
print(f"Prediction: {model.predict([[75, 3]])}") # y
```

Output:

```
Intercept: 0.7920792079206933
Coefficient/Slope: [1.98019802 0.1980198 ]
Prediction: [149.9009901]
```

ow let us see how the above fitted house price prediction model looks like in a plot.

Tutorial 7.3: To visualize the fitted line in *Tutorial 7.2* and the data points in a scatter plot, is as follows:

```
import matplotlib.pyplot as plt
# Extract the x and y values from the data
x_values = [row[0] for row in x]
y_values = y
# Plot the data points as a scatter plot
plt.scatter(x_values, y_values, color="blue", label="Data points")
# Plot the fitted line as a line plot
plt.plot(x_values, model.predict(x), color="red", label="Fitted linear regression model")
# Add some labels and a legend
plt.xlabel("Size of the house")
plt.ylabel("Price of the house")
plt.legend()
```

```
13. plt.savefig('fitting_models_to_independent_data.jpg',dpi
    =600,bbox_inches='tight') # Show the figure
14. plt.show() # Show the plot
```

Output:

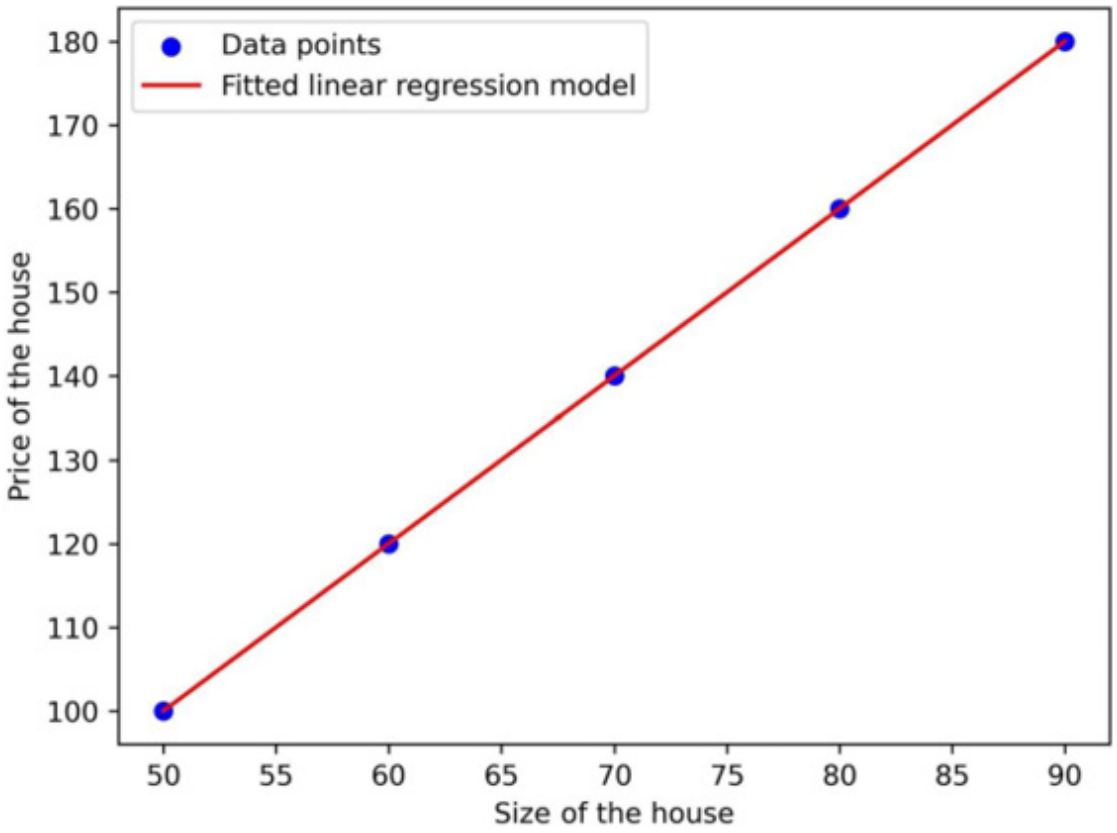

Figure 7.2: *Plot fitting size of house and price of house*

Linear regression is a suitable method for analyzing the relationship between a numerical outcome variable and one or more numerical or categorical characteristics. It is best used for data that exhibit a linear trend, where the change in the dependent variable is proportional to the change in the independent variables. If the data is non-linear as shown in *Figure 7.3*, linear regression may not be the most appropriate method, logistic regression, neural network and other algorithms may be more suitable. Linear regression is not suitable for data that follows a curved pattern, such as an exponential or logarithmic function, as it will not be able

to capture the true relationship and will produce a poor fit.

Tutorial 7.4: To show a scatter plot where data follow curved pattern, is as follows:

```
1. import numpy as np
2. import matplotlib.pyplot as plt
3. # Some data that follows a curved pattern
4. x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
5. y = np.sin(x)
6. # Plot the data as a scatter plot
7. plt.scatter(x, y, color='blue', label='Data')
8. # Fit a polynomial curve to the data
9. p = np.polyfit(x, y, 6)
10. y_fit = np.polyval(p, x)
11. # Plot the curve as a red line
12. plt.plot(x, y_fit, color='red', label='Curve')
13. # Add some labels and a legend
14. plt.xlabel('X')
15. plt.ylabel('Y')
16. plt.legend()
17. # Save the figure
18. plt.savefig('scatter_curve.png', dpi=600, bbox_inches='tight')
19. plt.show()
```

Output:

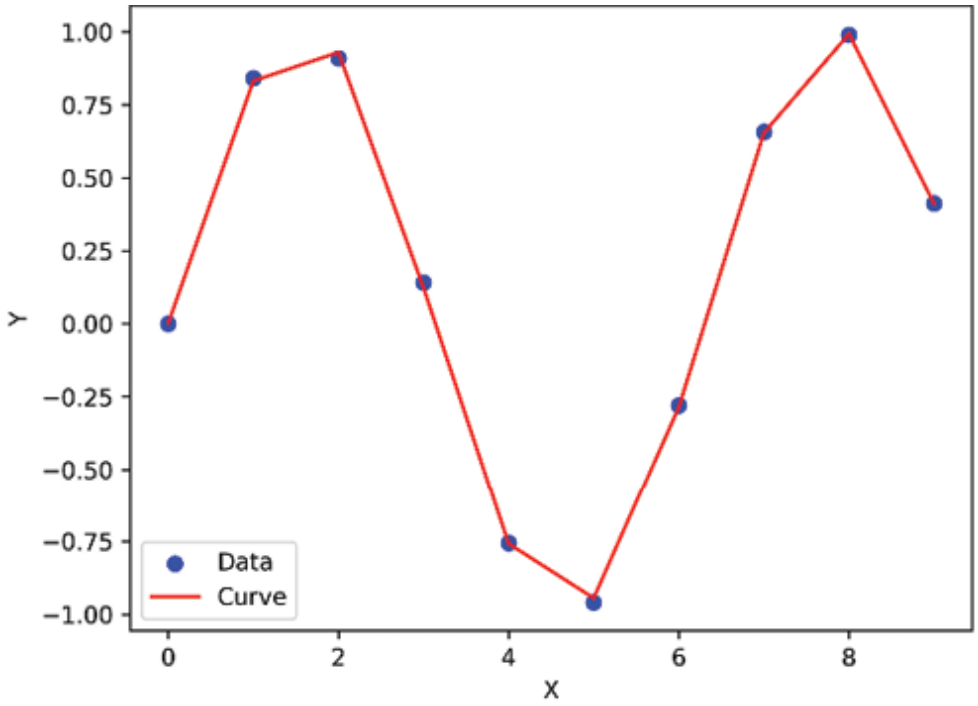

Figure 7.3: *Plot where X and Y data form a curved pattern line*

Therefore, it is important to check the assumptions of linear regression before applying it to the data, such as linearity, normality, homoscedasticity, and independence. Linearity can be easily viewed by plotting the data and looking for a linear pattern as shown in *Figure 7.4*.

Tutorial 7.5: To implement viewing of the linearity (linear pattern) in the data by plotting the data in a scatterplot, as follows:

```
1. import matplotlib.pyplot as plt
2. # Define the x and y variables
3. x = [1, 2, 3, 4, 5, 6, 7, 8]
4. y = [2, 4, 6, 8, 10, 12, 14, 16]
5. # Create a scatter plot
6. plt.scatter(x, y, color="red", marker="o")
7. # Add labels and title
8. plt.xlabel("x")
9. plt.ylabel("y")
10. plt.title("Linear relationship between x and y")
```

```
# Save the figure
plt.savefig('linearity.png', dpi=600, bbox_inches='tight')
plt.show()
```

Output:

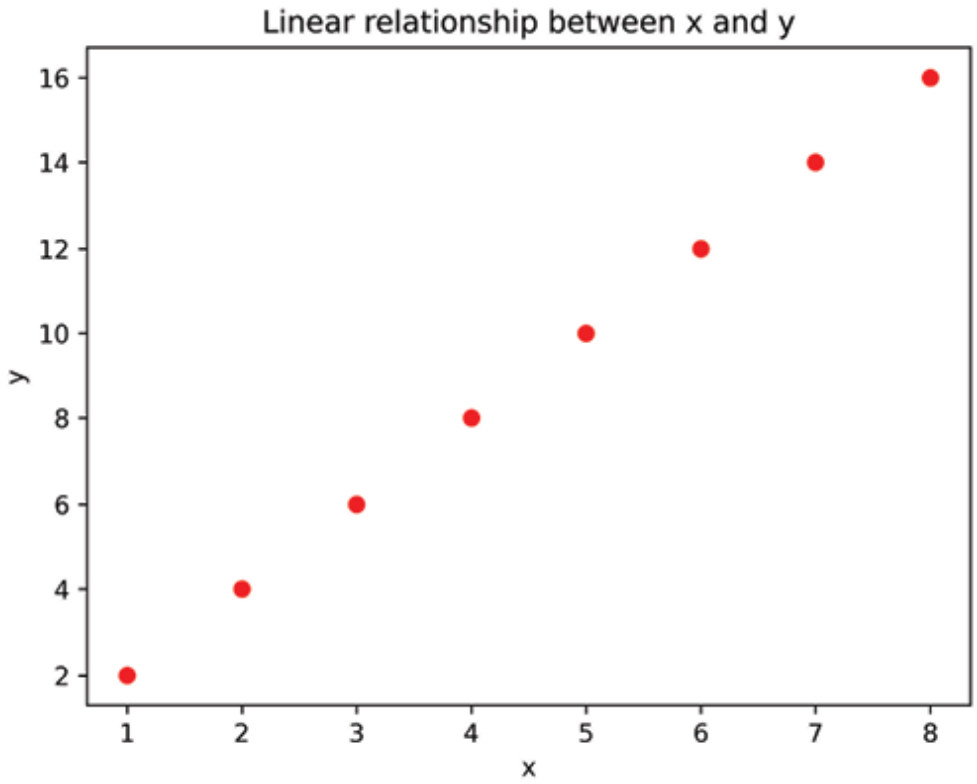

Figure 7.4: *Plot showing linearity (linear pattern) in the data*

It is also important that the residuals (the differences between the observed and predicted values) are normally distributed, have equal variances (homoscedasticity), and are independent of each other.

Tutorial 7.6: To check the normality of data, is as follows:

```
import matplotlib.pyplot as plt
import statsmodels.api as sm
# Define data
x = [1, 2, 3, 4, 5, 6, 7, 8]
y = [2, 4, 6, 8, 10, 12, 14, 16]
# Fit a linear regression model using OLS
```

```
model = sm.OLS(y, x).fit() # Create and fit an OLS objec
t
# Get the predicted values
y_pred = model.predict()
# Calculate the residuals
residuals = y - y_pred
# Plot the residuals
plt.scatter(y_pred, residuals, alpha=0.5)
plt.title('Residual Plot')
plt.xlabel('Predicted values')
plt.ylabel('Residuals')
# Save the figure
plt.savefig('normality.png', dpi=600, bbox_inches='tight'
)
plt.show()
```

sm.OLS() from the **statsmodels** module that performs **ordinary least squares (OLS)** regression, which is a method of finding the best-fitting linear relationship between a dependent variable and one or more independent variables.

The **output** is *Figure 7.5, it* does not fulfill the normality test or indicate that the residuals are normally distributed. It is a perfect fit, where the predicted values match exactly the observed values, and the residuals are all zero as follows:

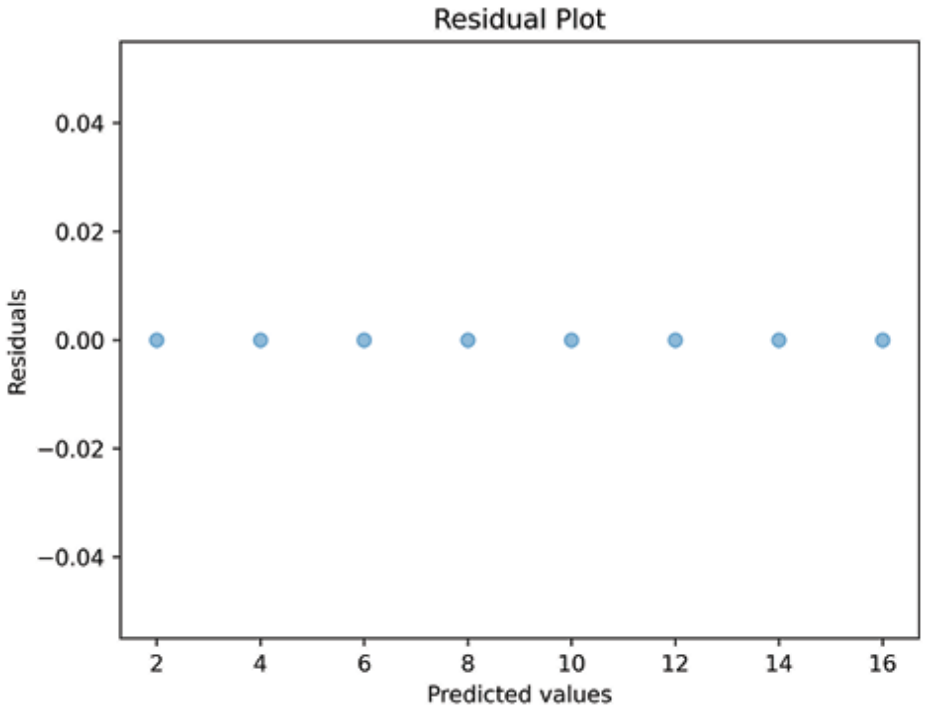

Figure 7.5: *Plot to view the normality in the data*

Further to check homoscedasticity create a scatter plot of the residuals and the predicted values to visually check if the residuals have constant variance at every level of the independent variables. Where independence means that the error for one observation does not affect the error for another observation, and is more useful to see for time-series data.

Logistic regression

Logistic regression is a type of statistical model that estimates the probability of an event occurring based on a given set of independent variables. It is often used for classification and predictive analytics, such as predicting whether an email is spam or not, or whether a customer will default on a loan or not. Logistic regression predicts the probability of an event or outcome using a set of predictor variables based on the concept of a logistic (sigmoid) function mapping a linear combination into a probability score between 0 and 1. Here, the predicted probability can

be used to classify the observation into one of the categories by choosing a cutoff value. For example, if the probability is greater than 0.5, the observation is classified as a success, otherwise it is classified as a failure.

For example, a simple example of logistic regression is to predict whether a student will pass an exam based on the number of hours they studied. Suppose we have the following data:

Hours studied	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
Passed	0	0	0	0	0	1	1	1	1	1

Table 7.1: *Hours studied and student exam result*

We can fit a logistic regression model to this data, using hours studied as the independent variable and passed as the dependent variable.

Before moving to the tutorials let us look at the syntax for implementing logistic regression with **sklearn**, which is as follows:

```
# Import logistic regression
from sklearn.linear_model import LogisticRegression
# Create a logistic regression model
logistic,_regression = LogisticRegression()
# Train the model
logistic_regression.fit(X_train, y_train)
```

Tutorial 7.7: To implement logistic regression based on above example, to predict whether a student will pass an exam based on the number of hours they studied, is as follows:

```
import numpy as np
import pandas as pd
# Import libraries from sklearn for logistic regression pr
ediction
from sklearn.linear_model import LogisticRegression
```

```
# Create the data
data = {"Hours studied": [0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5],
        "Passed": [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]}
df = pd.DataFrame(data)
# Define the independent and dependent variables
X = df["Hours studied"].values.reshape(-1, 1)
y = df["Passed"].values
# Fit the logistic regression model
model = LogisticRegression()
model.fit(X, y)
# Predict the probabilities for different values of hours studied
x_new = np.linspace(0, 6, 100).reshape(-1, 1)
y_new = model.predict_proba(x_new)[:, 1]
# Take the number of hours 3.76 to predict the probability of passing
x_fixed = 3.76
# Predict the probability of passing for the fixed number of hours
y_fixed = model.predict_proba([[x_fixed]])[0, 1]
# Print the fixed number of hours and the predicted probability
print(f"The fixed number of hours : {x_fixed:.2f}")
print(f"The predicted probability of passing : {y_fixed:.2f}")
```

Output:

```
The fixed number of hours : 3.76
The predicted probability of passing : 0.81
```

Tutorial 7.8: To visualize *Tutorial 7.7,* logistic regression model to predict whether a student will pass an exam based on the number of hours they studied in a plot is as follows:

```
import matplotlib.pyplot as plt
```

```
# Plot the data and the logistic regression curve
plt.scatter(X, y, color="blue", label="Data")
plt.plot(x_new, y_new, color="red", label="Logistic regression model")
plt.xlabel("Hours studied")
plt.ylabel("Probability of passing")
plt.legend()
# Show the figure
plt.savefig('student_reasult_prediction_model.jpg',dpi=600,bbox_inches='tight')
plt.show()
```

Output:

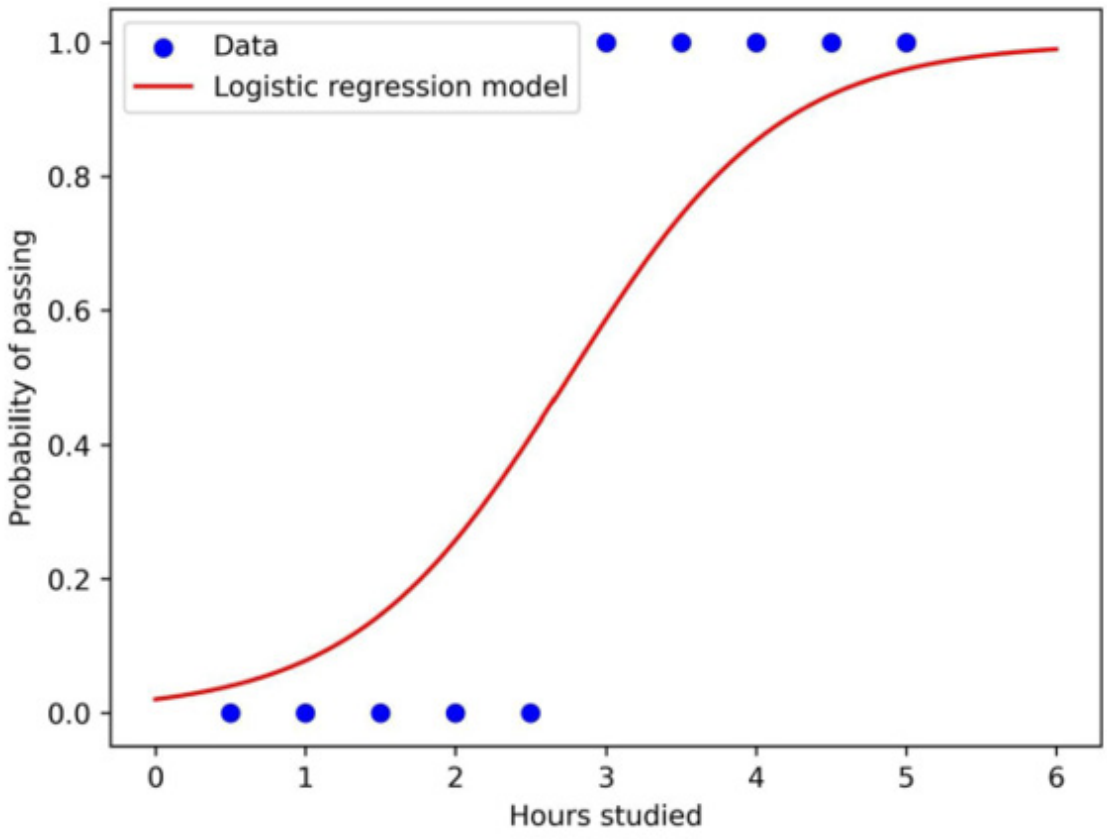

Figure 7.6. *Plot of fitted logistic regression model for prediction of student score*

Figure 7.6. shows that the probability of passing the final

exam increases as the number of hours studied increases, and that the logistic regression curve captures this trend well.

Fitting models to dependent data

Dependent data refers to related data points, such as repeated measurements on the same subject, clustered measurements from the same group, or spatial measurements from the same location. When fitting models to dependent data, it is important to account for the correlation structure among the data points. This can affect the estimation of the model parameters and the inference of the model effects. For example, fitting models to dependent data is to analyze the blood pressure of patients over time, who are assigned to different treatments. The blood pressure measurements of the same patient are likely to be correlated, and the patients may have different baseline blood pressure levels.

Linear mixed effect model

Linear Mixed-Effects Models (LMMs) are statistical models that can handle dependent data, such as data from longitudinal, multilevel, hierarchical, or correlated studies. They allow for both fixed and random effects. Fixed effects are the effects of variables that are assumed to have a constant effect on the outcome variable, while random effects are the effects of variables that have a varying effect on the outcome variable across groups or individuals. For example, suppose we have a data set of blood pressure measurements from 20 patients who are randomly assigned to one of two treatments: A or B. Blood pressure is measured at four time points: baseline, one month, two months, and three months. We can then fit a linear mixed effects model that predicts blood pressure based on treatment, time, and the interaction between them, while

accounting for correlation within each patient.

Tutorial 7.9: To implement linear mixed effect model to predict blood pressure from 20 patients, as follows:

```
import statsmodels.api as sm
# Generate some dummy data
import numpy as np
np.random.seed(50)
n_patients = 10 # Number of patients
n_obs = 5 # Number of observations per patient
x = np.random.randn(n_patients * n_obs) # Covariate
patient = np.repeat(np.arange(n_patients), n_obs) # Patient ID
bp = 100 + 5 * x + 10 * np.random.randn(n_patients * n_obs) # Blood pressure
# Create a data frame
import pandas as pd
df = pd.DataFrame({"bp": bp, "x": x, "patient": patient})
# Fit a linear mixed effect model with a random intercept for each patient
model = sm.MixedLM.from_formula("bp ~ x", groups="patient", data=df)
result = model.fit()
# Print the summary
print(result.summary())
```

Here we used **statsmodels** package, which provides a **MixedLM** class for fitting and analyzing mixed effect models.

Output:

```
        Mixed Linear Model Regression Results
==========================================================
Model:            MixedLM Dependent Variable: bp
```

```
No. Observations: 50      Method:            REML
No. Groups:       10      Scale:             132.8671
Min. group size:  5       Log-Likelihood:    -189.7517
Max. group size:  5       Converged:         Yes
Mean group size:  5.0
-----------------------------------------------------
            Coef.  Std.Err.   z    P>|z| [0.025  0.975]
-----------------------------------------------------
Intercept   99.960    1.711 58.427 0.000 96.607 103.314
x            4.021    1.686  2.384 0.017  0.716   7.326
patient Var  2.450    1.345
=====================================================
```

Output shows a linear mixed effect model with a random intercept for each patient, using total 50 observations from 10 patients. The model estimates a fixed intercept of 99.960, a fixed slope of 4.021, and a random intercept variance of 2.450 for each patient. The p-value for the slope is 0.017, which means that it is statistically significant at the 5% level. This implies that there is a positive linear relationship between the covariate x and the blood pressure bp, after accounting for the patient-level variability.

Similarly for fitting dependent data machine learning algorithms like logistic mixed-effects, K-nearest neighbors, multilevel logistic regression, marginal logistic regression, marginal linear regression can also be used.

Decision tree

Decision tree is a way of making decisions based on some data, they are used for both classification and regression problems. It looks like a tree with branches and leaves. Each branch represents a choice or a condition, and each leaf represents an outcome or a result. For example,

suppose you want to decide whether to play tennis or not based on the weather, if the weather is nice and sunny, you want to play tennis, if not, you do not want to play tennis. The decision tree works by starting with the root node, which is the top node. The root node asks a question about the data, such as **Is it sunny?** If the answer is yes, follow the branch to the right. If the answer is no, you follow the branch to the left. You keep doing this until you reach a leaf node that tells you the final decision, such as **Play tennis** or **Do not play tennis**.

Before moving to the tutorials let us look at the syntax for implementing decision tree with **sklearn**, which is as follows:

```
# Import decision tree
from sklearn.tree import DecisionTreeClassifier
# Create a decision tree classifier
tree = DecisionTreeClassifier()
# Train the classifier
tree.fit(X_train, y_train)
```

Tutorial 7.10: To implement a decision tree algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high is as follows:

```
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
# Read the data
data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
# Separate the features and the target
X = data.drop("blood_pressure", axis=1)
y = data["blood_pressure"]
# Encode the categorical features
X["gender"] = X["gender"].map({"M": 0, "F": 1})
```

```
# Build and train the decision tree
tree = DecisionTreeClassifier()
tree.fit(X, y)
```

Tutorial 7.11: To view graphical representation of the above fitted decision tree (*Tutorial 7.10*), showing the features, thresholds, impurity, and class labels at each node, is as follows:

```
import matplotlib.pyplot as plt
# Import the plot_tree function from the sklearn.tree module
from sklearn.tree import plot_tree
# Plot the decision tree
plt.figure(figsize=(10, 8))
# Fill the nodes with colors, round the corners, and add feature and class names
plot_tree(tree, filled=True, rounded=True, feature_names=X.columns, class_names=
["Low", "Normal", "High"], fontsize=12)
# Show the figure
plt.savefig('decision_tree.jpg',dpi=600,bbox_inches='tight')
plt.show()
```

Output:

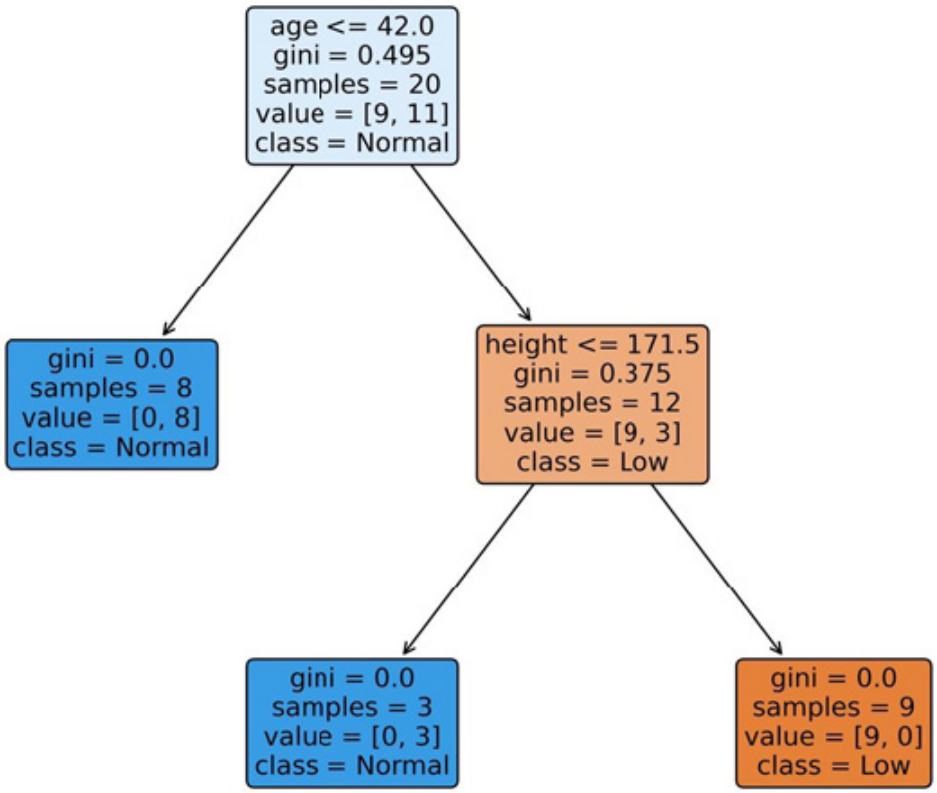

Figure 7.7: *Fitted decision tree plot with features, thresholds, impurity, and class labels at each node*

It is often a better idea to separate dependent and independent variables and split the dataset into train and test split before fitting the model. Independent data are the features or variables that are used as input to the model, and dependent data are the target or outcome that is predicted by the model. Splitting data into train test split is important because it allows us to evaluate the performance of the model on unseen data and avoid overfitting or underfitting. From the split, train set is used to fit or train the model and test set is used for evaluation of the model.

Tutorial 7.12: To implement decision tree by including the separation of dependent and independent variables, train test split and then fitting data on train set, based on *Tutorial 7.10* is as follows:

```
1. import pandas as pd
2. from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.model_selection import train_test_split
# Import the accuracy_score function
from sklearn.metrics import accuracy_score
# Read the data
data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
# Separate the features and the target
X = data.drop("blood_pressure", axis=1) # independent variables
y = data["blood_pressure"] # dependent variable
# Encode the categorical features
X["gender"] = X["gender"].map({"M": 0, "F": 1})
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Build and train the decision tree on the training set
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)
# Further test set can be used to evaluate the model
# Predict the values for the test set
y_pred = tree.predict(X_test) # Get the predicted values for the test data
# Calculate the accuracy score on the test set
accuracy = accuracy_score(y_test, y_pred) # Compare the predicted values with the actual values
# Print the accuracy score
print("Accuracy of the decision tree model on the test set :", accuracy)
```

After fitting the model on the training set, to use the remaining test set for evaluation of fitted model you need to import the **accuracy_score()** from the **sklearn.metrics**

module. Then use the **predict()** of the model on the test set to get the predicted values for the test data. Compare the predicted values with the actual values in the test set using the **accuracy_score()**, which returns a fraction of correct predictions. Finally print the accuracy score to see how well the model performs on the test data. More of this is discussed in the *Model selection and evaluation*.

Output:

```
1. Accuracy of the decision tree model on the test set : 1.0
```

This accuracy is quite high because we only have 20 data points in this dataset. Once we have adequate data, the above script will present more realistic results.

Random forest

Random forest is an ensemble learning method that combines multiple decision trees to make predictions. It is highly accurate and robust, making it a popular choice for a variety of tasks, including classification and regression, and other tasks that work by constructing a large number of decision trees at training time. Random forest works by building individual trees and then averaging the predictions of all the trees. To prevent overfitting, each tree is trained on a random subset of the training data and uses a random subset of the features. The random forest predicts by averaging the predictions of all the trees after building them. Averaging reduces prediction variance and improves accuracy.

For example, you have a large dataset of student data, including information about their grades, attendance, and extracurricular activities. As a teacher, you can use random forest to predict which students are most likely to pass their exams. To build a model, you would train a group of decision trees on different subsets of your data. Each tree would use a random subset of the features to make its predictions. After training all of the trees, you would

average their predictions to get your final result. This is like having a group of experts who each look at different pieces of information about your students. Each expert is like a decision tree, and they all make predictions about whether each student will pass or fail. After all the experts have made their predictions, you take an average of all the expert answers to give you the most likely prediction for each student.

Before moving to the tutorials let us look at the syntax for implementing random forest classifier with sklearn, which is as follows:

```
# Import RandomForestClassifier
from sklearn.ensemble import RandomForestClassifier
# Create a Random Forest classifier
rf = RandomForestClassifier()
# Train the classifier
rf.fit(X_train, y_train)
```

Tutorial 7.13. To implement a random forest algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high is as follows:

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
# Read the data
data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
# Separate the features and the target
X = data.drop("blood_pressure", axis=1) # independent variables
y = data["blood_pressure"] # dependent variable
# Encode the categorical features
X["gender"] = X["gender"].map({"M": 0, "F": 1})
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test
```

```
_size=0.2, random_state=42)
# Create a Random Forest classifier
rf = RandomForestClassifier()
# Train the classifier
rf.fit(X_train, y_train)
```

Tutorial 7.14: To evaluate *Tutorial 7.13,* fitted random forest classifier on the test set of data append these lines of code at the end of *Tutorial 7.13*:

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Further test set can be used to evaluate the model
# Predict the values for the test set
y_pred = tree.predict(X_test) # Get the predicted values for the test data
# Calculate the accuracy score on the test set
accuracy = accuracy_score(y_test, y_pred) # Compare the predicted values with the actual values
# Print the accuracy score
print("Accuracy of the Random Forest classifier model on the test set :", accuracy)
```

Support vector machine

Support Vector Machines (SVMs) are a type of supervised machine learning algorithm used for classification and regression tasks. They find a hyperplane that separates data points of different classes, maximizing the margin between them. SVMs map data points into a high-dimensional space to make separation easier. A kernel function is used to map data points and measure their similarity in high-dimensional space. SVMs then find the hyperplane that maximizes the margin between the two classes. SVMs are versatile. They can be used for classification, regression, and anomaly detection. They are

particularly well-suited for tasks where the data is nonlinear or has high dimensionality. They are also quite resilient to noise and outliers.

For example, imagine you are a doctor trying to diagnose a patient with a certain disease. patient records that include information about their symptoms, medical history, and blood test results. To predict whether a new patient has the disease or not, you can use SVM to build a model. First, train the SVM on the dataset of patient records. SVM would identify the most important features of the data to distinguish between patients with and without the disease. Then, it could predict whether a new patient has the disease based on their symptoms, medical history, and blood test results.

Before moving to the tutorials let us look at the syntax for implementing support vector classifier from SVM with sklearn, which is as follows:

```
# Import Support vector classifier from SVM
from sklearn.svm import SVC
# Create a Support Vector Classifier object
svm = SVC()
# Train the classifier
svm.fit(X_train, y_train)
```

Tutorial 7.15. To implement SVM, support vector classifier algorithm on patient data to classify the blood pressure of 20 patients into low, normal, high and evaluate the result is as follows:

```
import pandas as pd
# Import the SVC class
from sklearn.svm import SVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
# Read the data
```

```
data = pd.read_csv("/workspaces/ImplementingStatisticsWithPython/data/chapter7/patient_data.csv")
# Separate the features and the target
X = data.drop("blood_pressure", axis=1) # independent variables
y = data["blood_pressure"] # dependent variable
# Encode the categorical features
X["gender"] = X["gender"].map({"M": 0, "F": 1})
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create an SVM classifier
svm = SVC(kernel="rbf", C=1, gamma=0.1) # You can change the parameters as you wish
# Train the classifier
svm.fit(X_train, y_train)
# Predict the values for the test set
y_pred = svm.predict(X_test) # Get the predicted values for the test data
# Calculate the accuracy score on the test set
accuracy = accuracy_score(y_test, y_pred) # Compare the predicted values with the actual values
# Print the accuracy score
print("Accuracy of the SVM classifier model on the test set :", accuracy)
```

The output of this will be a trained model and accuracy score of classifiers.

K-nearest neighbor

K-Nearest Neighbor (KNN) is a machine learning algorithm used for classification and regression. It finds the k nearest neighbors of a new data point in the training data

and uses the majority class of those neighbors to classify the new data point. KNN is useful when the data is not linearly separable, meaning that there is no clear boundary between different classes or outcomes. KNN is useful when dealing with data that has many features or dimensions because it makes no assumptions about the distribution or structure of the data. However, it can be slow and memory-intensive since it must store and compare all the training data for each prediction.

A simpler example to explain it is, suppose you want to predict the color of a shirt based on its size and price. The training data consists of ten shirts, each labeled as either red or blue. To classify a new shirt, we need to find the k closest shirts in the training data, where k is a number chosen by us. For example, if $k = 3$, we look for the 3 nearest shirts based on the difference between their size and price. Then, we count how many shirts of each color are among the 3 nearest neighbors, and assign the most frequent color to the new shirt. For example, if 2 of the 3 nearest neighbors are red, and 1 is blue, we predict that the new shirt is red.

Let us see a tutorial to predict the type of flower based on its features, such as petal length, petal width, sepal length, and sepal width. The training data consists of 150 flowers, each labeled as one of three types: Iris setosa, Iris versicolor, or Iris virginica. The number of k is chosen by us. For instance, if $k = 5$, we look for the 5 nearest flowers based on the Euclidean distance between their features. We count the number of flowers of each type among the 5 nearest neighbors and assign the most frequent type to the new flower. For instance, if 3 out of the 5 nearest neighbors are Iris versicolor and 2 are Iris virginica, we predict that the new flower is Iris versicolor.

Tutorial 7.16: To implement KNN on iris dataset to predict the type of flower based on its features, such as petal

length, petal width, sepal length, and sepal width and also evaluate the result, is as follows:

```
# Load the Iris dataset
from sklearn.datasets import load_iris
 # Import the KNeighborsClassifier class
from sklearn.neighbors import KNeighborsClassifier
# Import train_test_split for data splitting
from sklearn.model_selection import train_test_split
# Import accuracy_score for evaluating model performa
nce
from sklearn.metrics import accuracy_score
# Load the Iris dataset
iris = load_iris()
# Separate the features and the target variable
X = iris.data  # Features (sepal length, sepal width, petal
 length, petal width)
y = iris.target  # Target variable (species: Iris-
setosa, Iris-versicolor, Iris-virginica)
# Encode categorical features (if any)
# No categorical features in the Iris dataset
# Split the data into training (90%) and test sets (10%)
X_train, X_test, y_train, y_test = train_test_split(X, y, test
_size=0.1, random_state=42)
# Create a KNeighborsClassifier object
knn = KNeighborsClassifier(n_neighbors=5)  # Set num
ber of neighbors to 5
# Train the classifier
knn.fit(X_train, y_train)
# Make predictions on the test data
y_pred = knn.predict(X_test)
# Evaluate the model's performance using accuracy
```

```
accuracy = accuracy_score(y_test, y_pred)
# Print the accuracy score
print("Accuracy of the KNN classifier on the test set :", accuracy)
```

Output:

```
Accuracy of the KNN classifier on the test set : 1.0
```

Model selection and evaluation

Model selection and evaluation methods are techniques used to measure the performance and quality of machine learning models. Supervised learning methods commonly use evaluation metrics such as accuracy, precision, recall, F1-score, mean squared error, mean absolute error, and area under the curve. Unsupervised learning methods commonly use evaluation metrics such as silhouette score, Davies-Bouldin index, Calinski-Harabasz index, and adjusted Rand index.

Evaluation metrices and model selection for supervised

As mentioned above and to summarize choosing a single candidate machine learning model for a predictive modeling challenge is known as model selection. Performance, complexity, interpretability, and resource requirements are some examples of selection criteria. As mentioned, accuracy, precision, recall, F1-score, and area under the curve are highly relevant to evaluate classifier result. **Mean Absolute Error (MAE), Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-Squared (R2)** are useful for evaluating prediction models. *Tutorial 7.30* demonstrates for each section, shows how to use some common model selection and evaluation techniques for supervised learning using the scikit-learn library.

Tutorial 7.30: To implement a tutorial that illustrates

model selection and evaluation in supervised machine learning using iris data, is as follows:

To begin, we need to import modules and load the iris dataset. This dataset contains 150 samples of three different types of iris flowers, each with four features: sepal length, sepal width, petal length, and petal width. Our goal is to construct a classifier that can predict the species of a new flower based on its features as follows:

```
import numpy as np # For numerical operations
import pandas as pd # For data manipulation and analysis
import matplotlib.pyplot as plt # For data visualization
from sklearn.datasets import load_iris # For loading the iris dataset
from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV, RandomizedSearchCV # For splitting data, cross-
validation, and hyperparameter tuning
from sklearn.linear_model import LogisticRegression # For logistic regression model
from sklearn.tree import DecisionTreeClassifier # For decision tree model
from sklearn.svm import SVC # For support vector machine model
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report # For evaluating model performance
# Load dataset
iris = load_iris()
# Extract the features & labels as a numpy array
X = iris.data
y = iris.target
```

```
print(iris.feature_names)
print(iris.target_names)
```

Output:

```
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
['setosa' 'versicolor' 'virginica']
```

Continuing the *Tutorial 7.30* we will now split the data set into training and test sets. The data will be divided into 70% for training and 30% for testing. Additionally, we will set a random seed for reproducibility as follows:

```
# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

Output:

```
(105, 4) (105,)
(45, 4) (45,)
```

Now, we will define candidate models in *Tutorial 7.30* to compare, including logistic regression, decision tree, and support vector machine classifiers. Candidate model refers to a machine learning algorithm that is being considered or tested to solve a particular problem. We will use default values for hyperparameters. However, they can be adjusted to find the optimal solution as follows:

```
# Define candidate models
models = {
    'Logistic Regression': LogisticRegression(),
    'Decision Tree': DecisionTreeClassifier(),
    'Support Vector Machine': SVC()
}
```

To evaluate the performance of each model in *Tutorial 7.30,* we can use cross-validation. This technique involves

splitting the training data into k folds. One-fold is used as the validation set, and the rest is used as the training set. This process is repeated **k** times, and the average score across the k folds is reported. Cross-validation helps to reduce the variance of the estimate and avoid overfitting. In this case, we will use 5-fold cross-validation and accuracy as the scoring metric as follows:

```
# Evaluate models using cross-validation
scores = {}
for name, model in models.items():
    score = cross_val_score(model, X_train, y_train, cv=5, scoring='accuracy')
    scores[name] = np.mean(score)
    print(f'{name}: {np.mean(score):.3f} (+/- {np.std(score):.3f})')
```

Output:

```
Logistic Regression: 0.962 (+/- 0.036)
Decision Tree: 0.933 (+/- 0.023)
Support Vector Machine: 0.952 (+/- 0.043)
```

The logistic regression and support vector machine models have comparatively almost similar and high accuracy scores. However, the decision tree model has a slightly lower score, indicating overfitting and poor generalization. To better compare the results, a bar chart can be plotted as follows:

```
# Plot scores
plt.bar(scores.keys(), scores.values())
plt.ylabel('Accuracy')
plt.show()
```

Output:

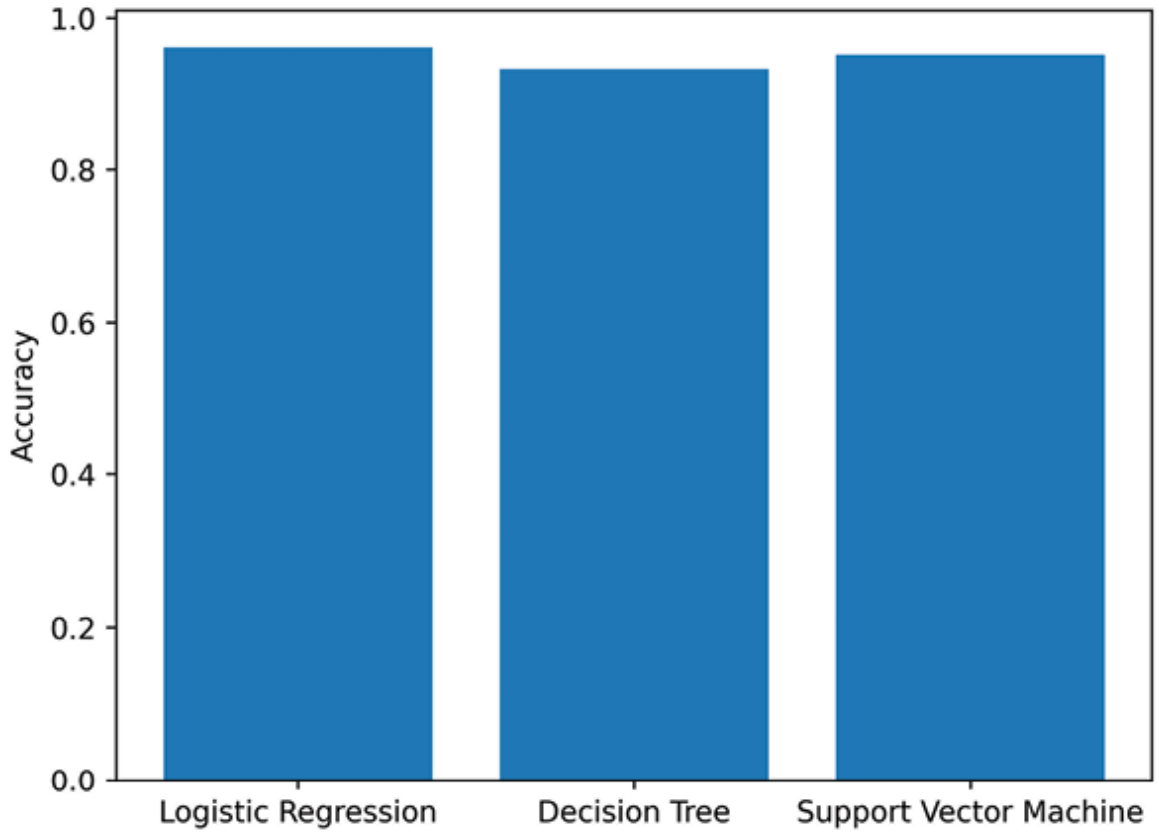

Figure 7.8: *Accuracy comparison of supervised algorithms on the Iris dataset*

As evaluation measures how well, the model performs on unseen data, such as a test set, by comparing its predictions to the actual results using various metrics. Now we evaluate each model's performance, use the testing set. Fit each model on the training set, make predictions on the testing set, and compare them with the true labels. Compute metrics such as accuracy, confusion matrix, and classification report. The confusion matrix shows the number of accurate and inaccurate predictions for each class, while the classification report presents the precision, recall, f1-score, and support for each class as follows:

```
# Evaluate models using testing set
for name, model in models.items():
    model.fit(X_train, y_train) # Fit model on training set
    y_pred = model.predict(X_test) # Predict on testing se
```

```
t
    acc = accuracy_score(y_test, y_pred) # Compute accuracy
    cm = confusion_matrix(y_test, y_pred) # Compute confusion matrix
    prec = precision_score(y_test, y_pred, average='weighted') # Compute precision
    recall = recall_score(y_test, y_pred, average='weighted') # Compute recall
    f1score = f1_score(y_test, y_pred, average='weighted') # Compute f1score
    print(f'\n{name}')
    print(f'Accuracy: {acc:.3f}')
    print(f'Precision: {prec:.3f}')
    print(f'Recall: {recall:.3f}')
    print(f'Confusion matrix:\n{cm}')
```

Output:

```
Logistic Regression
Accuracy: 1.000
Precision: 1.000
Recall: 1.000
Confusion matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
Decision Tree
Accuracy: 1.000
Precision: 1.000
Recall: 1.000
Confusion matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

```
Support Vector Machine
Accuracy: 1.000
Precision: 1.000
Recall: 1.000
Confusion matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

The logistic regression, decision tree, and support vector machine models all have the highest accuracy, precision, recall, and f1 score of 1.0. All of these and the confusion matrix indicate that all models have perfect predictions for all classes. Therefore, all models are equally effective for this classification problem. However, it is important to consider factors other than performance, such as complexity, interpretability, and resource requirements. For example, the logistic regression model is the simplest and most interpretable model. On the other hand, the support vector machine model and the decision tree model are the most complex and least interpretable models. The resource requirements for each model depend on the size and dimensionality of the data, the number and range of hyperparameters, and the available computing power. Therefore, the selection of the final model depends on the trade-off between these factors.

Semi-supervised and self-supervised learnings

Semi-supervised learning is a paradigm that combines both labeled and unlabeled data for training machine learning models. In this approach, we have a limited amount of labeled data (with ground truth labels) and a larger pool of unlabeled data. The goal is to leverage the unlabeled data to improve model performance. It bridges the gap between fully supervised (only labeled data) and unsupervised (no

labels) learning. Imagine you're building a spam email classifier. You have a small labeled dataset of spam and non-spam emails, but a vast number of unlabeled emails. By using semi-supervised learning, you can utilize the unlabeled emails to enhance the classifier's accuracy.

Self-supervised learning is a type of unsupervised learning where the model generates its own labels from the input data. Instead of relying on external annotations, the model creates its own supervision signal. Common self-supervised tasks include predicting missing parts of an input (e.g., masked language models) or learning representations by solving pretext tasks (e.g., word embeddings). Consider training a neural network to predict the missing word in a sentence. Given the sentence: **The cat chased the blank**, the model learns to predict the missing word **mouse**. Here, the model generates its own supervision by creating a masked input. Thus, the key difference lies in semi-supervised and self-supervised is the source of supervision.

- **Semi-supervised** uses a small amount of labeled data and a larger pool of unlabeled data.
    - **Use case**: When labeled data is scarce or expensive to obtain.
    - **Example**: Pretraining language models like BERT on large text corpora without explicit labels.
- S**elf-supervised learning** creates its own supervision signal from the input data.
    - **Use case**: When you have some labeled data but want to leverage additional unlabeled data.
    - **Example**: Enhancing image classification models by incorporating unlabeled images alongside labeled ones.

Semi-supervised techniques

Semi-supervised learning bridges the gap between fully supervised and unsupervised learning. It leverages both labeled and unlabeled data to improve model performance. Semi-supervised techniques allow us to make the most of limited labeled data by incorporating unlabeled examples. By combining these methods, we achieve better generalization and performance in real-world scenarios In this chapter, we explore three essential semi-supervised techniques which are **self-training**, **co-training**, and **graph-based methods**, each with a specific task or idea, along with examples to address or solve them.

- **Self-training**: Self-training is a simple yet effective approach. It starts with an initial model trained on the limited labeled data available. The model then predicts labels for the unlabeled data, and confident predictions are added to the training set as pseudo-labeled examples. The model is retrained using this augmented dataset, iteratively improving its performance. Suppose we have a sentiment analysis task with a small labeled dataset of movie reviews. We train an initial model on this data. Next, we apply the model to unlabeled reviews, predict their sentiments, and add the confident predictions to the training set. The model is retrained, and this process continues until convergence.
    - **Idea**: Iteratively label unlabeled data using model predictions.
    - **Example**: Train a classifier on labeled data, predict labels for unlabeled data, and add confident predictions to the labeled dataset.

Tutorial 7.32: To implement self-training classifier on Iris dataset, as follows:

```
from sklearn.semi_supervised import SelfTrainingClassifier
from sklearn.datasets import load_iris
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
# Load the Iris dataset (labeled data)
X, y = load_iris(return_X_y=True)
# Split data into labeled and unlabeled portions
X_labeled, X_unlabeled, y_labeled, y_unlabeled = train_test_split(X, y, test_size=0.8, random_state=42)
# Initialize a base classifier (e.g., logistic regression)
base_classifier = LogisticRegression()
# Create a self-training classifier
self_training_clf = SelfTrainingClassifier(base_classifier)
# Fit the model using labeled data
self_training_clf.fit(X_labeled, y_labeled)
# Predict on unlabeled data
y_pred_unlabeled = self_training_clf.predict(X_unlabeled)
# Print the original labels for the unlabeled data
print("Original labels for unlabeled data:")
print(y_unlabeled)
# Print the predictions
print("Predictions on unlabeled data:")
print(y_pred_unlabeled)
```

Output:

```
Original labels for unlabeled data:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0
 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
 2 0 1 2 0 2 2
 1 1 2 1 0 1 2 0 0 1 1 0 2 0 0 1 1 2 1 2 2 1 0 0 2 2 0 0 0 1
 2 0 2 2 0 1 1
 2 1 2 0 2 1 2 1 1]
Predictions on unlabeled data:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0 1 2 1 1 2 0 2 0 2 2 2 2 2 0 0
```

```
 0 0 1 0 0 2 1
 0 0 0 2 1 1 0 0 1 2 2 1 2 1 2 1 0 2 1 0 0 0 1 2 0 0 0 1 0 1
 2 0 1 2 0 2 2
 1 1 2 1 0 1 2 0 0 1 2 0 2 0 0 2 1 2 2 2 2 1 0 0 1 2 0 0 0 1
 2 0 2 2 0 1 1
 2 1 2 0 2 1 2 1 1]
```

The above outputs have few wrong predictions. Now, let us see the evaluation metrics.

Tutorial 7.33: To evaluate the trained self-training classifier performance using appropriate metrics (e.g., accuracy, F1-score, etc.), as follows:

```
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
# Assuming y_unlabeled_true contains true labels for unlabeled data
accuracy = accuracy_score(y_unlabeled, y_pred_unlabeled)
f1 = f1_score(y_unlabeled, y_pred_unlabeled, average='weighted')
precision = precision_score(y_unlabeled, y_pred_unlabeled, average='weighted')
recall = recall_score(y_unlabeled, y_pred_unlabeled, average='weighted')
print(f"Accuracy: {accuracy:.2f}")
print(f"F1-score: {f1:.2f}")
print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

Output:

```
Accuracy: 0.97
F1-score: 0.97
Precision: 0.97
Recall: 0.97
```

Here, we see an accuracy of 0.97 means that approximately

97% of the predictions were correct. F1-score of 0.97 suggests a good balance between precision and recall, where higher values indicate better performance. A precision of 0.97 means that 97% of the positive predictions were accurate. A recall of 0.97 indicates that 97% of the positive instances were correctly identified. Further calibration of the classifier is essential for better results. You can fine-tune hyperparameters or use techniques like Platt scaling or isotonic regression to improve calibration.

- **Co-training**: Co-training leverages multiple views of the data. It assumes that different features or representations can provide complementary information. Two or more classifiers are trained independently on different subsets of features or views. During training, they exchange their confident predictions on unlabeled data, reinforcing each other's learning. Consider a text classification problem where we have both textual content and associated metadata, for example, author, genre. We train one classifier on the text and another on the metadata. They exchange predictions on unlabeled data, improving their performance collectively.
    - **Idea**: Train multiple models on different views of data and combine their predictions.
    - **Example**: Train one model on text features and another on image features, then combine their predictions for a joint task.

Tutorial 7.34: To show and easy implementation of co-training with two views of data, on **UCImultifeature** dataset from **mvlearn.datasets,** as follows:

```
from mvlearn.semi_supervised import CTClassifier
from mvlearn.datasets import load_UCImultifeature
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
```

```
data, labels = load_UCImultifeature(select_labeled=
[0,1])
X1 = data[0]  # Text view
X2 = data[1]  # Metadata view
X1_train, X1_test, X2_train, X2_test, l_train, l_test = trai
n_test_split(X1, X2, labels)
# Co-
training with two views of data and 2 estimator types
estimator1 = LogisticRegression()
estimator2 = RandomForestClassifier()
ctc = CTClassifier(estimator1, estimator2, random_state
=1)
# Use different matrices for each view
ctc = ctc.fit([X1_train, X2_train], l_train)
preds = ctc.predict([X1_test, X2_test])
print("Accuracy: ", sum(preds==l_test) / len(preds))
```

This code snippet illustrates the application of co-training, a semi-supervised learning technique, using the **CTClassifier** from **mvlearn.semi_supervised**. Initially, a multi-view dataset is loaded, focusing on two specified classes. The dataset is divided into two views: text and metadata. Following this, the data is split into training and testing sets. Two distinct classifiers, logistic regression and random forest, are instantiated. These classifiers are then incorporated into the **CTClassifier**. After training on the training data from both views, the model predicts labels for the test data. Finally, the accuracy of the co-training model on the test data is computed and displayed. Output will display the accuracy of the model as follows:

- **Graph-based methods**: Graph-based methods exploit the inherent structure in the data. They construct a graph where nodes represent instances (labeled and unlabeled), and edges encode similarity or relationships. Label propagation or graph-based regularization is then

used to propagate labels across the graph, benefiting from both labeled and unlabeled data. In a recommendation system, users and items can be represented as nodes in a graph. Labeled interactions (e.g., user-item ratings) provide initial labels. Unlabeled interactions contribute to label propagation, enhancing recommendations as follows:

  - **Idea**: Leverage data connectivity (e.g., graph Laplacians) for label propagation.
  - **Example**: Construct a graph where nodes represent data points, and edges represent similarity. Propagate labels across the graph.

Self-supervised techniques

Self-supervised learning techniques empower models to learn from unlabeled data, reducing the reliance on expensive labeled datasets. These methods exploit inherent structures within the data itself to create meaningful training signals. In this chapter, we delve into three essential self-supervised techniques: **word embeddings**, **masked language models**, and **language models**.

- **Word embeddings**: A word embedding is a representation of a word as a real-valued vector. These vectors encode semantic meaning, allowing similar words to be close in vector space. Word embeddings are crucial for various **Natural Language Processing (NLP)** tasks. They can be obtained using techniques like neural networks, dimensionality reduction, and probabilistic models. For instance, **Word2Vec** and **GloVe** are popular methods for generating word embeddings. Let us consider an example, suppose we have a corpus of text. Word embeddings capture relationships between words. For

instance, the vectors for king and queen should be similar because they share a semantic relationship.

- **Idea**: Pretrained word representations.
- **Use**: Initializing downstream models, for example natural language processing tasks.

Tutorial 7.35: To implement word embeddings using self-supervised task using **Word2Vec** method, as follows:

```
# Install Gensim and import word2vec for word embeddings
import gensim
from gensim.models import Word2Vec
# Example sentences
sentences = [
    ["I", "love", "deep", "learning"],
    ["deep", "learning", "is", "fun"],
    ["machine", "learning", "is", "easy"],
    ["deep", "learning", "is", "hard"],
    # Add more sentences, embeding changes with new words...
]
# Train Word2Vec model
model = Word2Vec(sentences, vector_size=10, window=5, min_count=1, sg=1)
# Get word embeddings
word_vectors = model.wv
# Example: Get embedding for the each word in sentence "I love deep learning"
print("Embedding for 'I':", word_vectors["I"])
print("Embedding for 'love':", word_vectors["love"])
print("Embedding for 'deep':", word_vectors["deep"])
print("Embedding for 'learning':", word_vectors["learning"])
```

Output:

1. Embedding for 'I': [-0.00856557 0.02826563 0.0540142
   9
   0.07052656 -0.05703121 0.0185882
2. 0.06088864 -0.04798051 -0.03107261 0.0679763]
3. Embedding for 'love': [0.05455794 0.08345953 -0.0145
   3741
   -0.09208143 0.04370552 0.00571785
4. 0.07441908 -0.00813283 -0.02638414 -0.08753009]
5. Embedding for 'deep': [0.07311766 0.05070262 0.067
   57693
   0.00762866 0.06350891 -0.03405366
6. -0.00946401 0.05768573 -0.07521638 -0.03936104]
7. Embedding for 'learning': [-0.00536227 0.00236431
   0.0510335   0.09009273 -0.0930295 -0.07116809
8. 0.06458873 0.08972988 -0.05015428 -0.03763372]

- **Masked Language Models (MLM**): MLM is a powerful self-supervised technique used by models like **Bidirectional Encoder Representations from Transformers (BERT).** In MLM, some tokens in an input sequence are m asked, and the model learns to predict these masked tokens based on context. It considers both preceding and following tokens, making it bidirectional. Given the sentence: **The cat sat on the [MASK]**. The model predicts the masked token, which could be **mat**, **chair**, or any other valid word based on context as follows:
  - **Idea**: Unidirectional pretrained language representations.
  - **Use**: Full downstream model initialization for various language understanding tasks.
- **Language models**: A language model is a probabilistic model of natural language. It estimates the likelihood of a sequence of words. Large language models, such as GPT-4 and ELMo, combine neural

networks and transformers. They have superseded earlier models like n-gram language models. These models are useful for various NLP tasks, including speech recognition, machine translation, and information retrieval. Imagine a language model trained on a large corpus of text. Given a partial sentence, it predicts the most likely next word. For instance, if the input is **The sun is shining**, the model might predict **brightly** as follows:

- **Idea**: Bidirectional pretrained language representations.
- **Use**: Full downstream model initialization for tasks like text classification and sentiment analysis.

Conclusion

In this chapter, we explored the basics and applications of statistical machine learning. Supervised machine learning is a powerful and versatile tool for data analysis and AI for labeled data. Knowing the type of problem, whether supervised or unsupervised, solves half the learning problems; the next step is to implement different models and algorithms. Once this is done, it is critical to evaluate and compare the performance of different models using techniques such as cross-validation, bias-variance trade-off, and learning curves. Some of the best known and most commonly used supervised machine learning techniques have been demonstrated. These techniques include decision trees, random forests, support vector machines, K-nearest neighbors, linear and logistic regression. We've also talked about semi-supervised and self-supervised, and techniques for implementing them. We have also mentioned the advantages and disadvantages of each approach, as well as some of the difficulties and unanswered questions in the field of machine learning.

Chapter 8, Unsupervised Machine Learning explores the other type of statistical machine learning, unsupervised machine learning.